Amendments to the Specification:

Please replace the paragraph on Page 2, lines 1-13, with the following amended paragraph:

The situation described above is not unique to surface modeling. Software engineering discipline faces similar problems. When modeling software engineering artifacts, a user can often be frustrated because control points provided to the user are closely tied to the representations' degree of freedom. In this vein, one may consider object-oriented (OO) design methodology for software development. An OO language, such as ~~Java~~ JAVA or C++, provides a user with a few control points for manipulating software artifacts. Inheritance lets a user extend a class. In that class extensions tend not to represent the only type of controls that a user wants, it is often very difficult, if not impossible, to make a simple change, such as adding logging feature to a set of classes. [14]. Accordingly, a need has been recognized in connection with providing a user with an infinitely malleable software artifact having no fixed structure or controls of its own. To this software artifact additional features or control points can then be added to obtain concrete controllable software artifacts.

Please replace the paragraph on Page 8, lines 5-17, with the following amended paragraph:

Herein there is also described the semantics of extension types, and several applications of extension types are given in the context of aspect-oriented software

development and design patterns. There is introduced the notion of parameterized extension types that provide more type-safe composition than is possible with pure extension types. There are also presented manners of modeling both multiple classification and dynamic classification using extension types (see [9]). (Particularly, whereas [9] is an early article showing multiple/dynamic classification, it hard-codes the methodology rather than use a more flexible system as contemplated herein). Finally, it will be shown how extension types can added to an ~~AspectJ~~ ASPECTJ framework to provide a notion sub-typing to classes and aspects (see [6]). (Particularly, [6] represents an early paper introducing ~~AspectJ~~ ASPECTJ. While aspects in ~~AspectJ~~ ASPECTJ are not first class types, it will be demonstrated herein that by using extension types, aspects can be configured as first class types.)

Please replace the paragraph on Page 9, lines 12-18, with the following amended paragraph:

Separation of concerns (SOC), a term introduced by Djisktra, refers to a software engineering concept for identifying, encapsulating, and manipulating different features (aspects, concerns, etc) of software artifacts so that one can organize and decompose software into manageable and comprehensible parts. [13][14] A class in OO languages is one kind of a concern. There are others kinds of concern that can cut across multiple classes, such as logging, printing, persistence, and display capabilities. ~~HyperJ~~ HYPERJ

and ~~AspectJ™~~ ASPECTJ are two different implementations of SOC in ~~Java~~ JAVA [13][6].

Please replace the paragraph bridging Pages 10-11 with the following amended paragraph:

~~HyperJ~~ HYPERJ uses concepts such as hyperslice and hyperspace. [13] A hyperspace is a "space of concerns" that is spanned by a set of "vectors of concerns". A set of "vector of concerns" is called a hyperslice of a hyperspace. A "basis concern" for a hyperspace is an independent set of concerns that span the hyperspace. A dimension of a hyperspace *H* is the number of elements (concerns) in a basis concern for *H*. A hypermodule *M* is a sub-hyperspace of a hyperspace *H* that consists of hyperslices along with operations for composing hyperslices. Hypermodules are building blocks, and are not, in general, complete, executable programs. A system is a hypermodule that is complete, and can therefore run independently. OO languages such as ~~Java~~ JAVA and C++ support what is termed as a "tyranny of dominant decomposition," or separation and encapsulation along only one dominant hyperslice. [13][14]. For instance, consider the class hierarchy of employees in a company as shown in Fig. 1. As shown, the class hierarchy contains at least two hyperslices that are tangled: (1) personnel hyperslice including employee name, identity, management hierarchy, etc., and (2) payroll hyperslice including salary, tax, and other related information. There is a poor separation of concerns in such a class design. One way to separate the personnel concern from

payroll concern is to create two class hierarchies: one for personnel department and another for payroll department. An Employee class hierarchy can then be constructed by appropriately composing the two hyperslices. ~~HyperJ~~ HYPERJ, for instance, proposes a set of composition operations on hyperslices. An interesting aspect of the ~~HyperJ~~ HYPERJ approach is that composition operations are separate from hyperslices. This allows one to construct new class hierarchies in a non-intrusive manner.

Please replace the paragraph bridging Pages 11-12 with the following amended paragraph:

~~AspectJ™~~ ASPECTJ is a general-purpose aspect-oriented programming (AOP) extension to ~~Java~~ JAVA. [6] In AspectJ, "aspects" modularize concerns that affect one or more classes. ~~AspectJ~~ ~~ASPECTJ~~ uses the notion of "join points" to insert new behavior in existing code. Join points are well-defined points in the execution of a program, which includes, reading or writing a field; calling or executing an exception handler, method or constructor. These join points are described by the "pointcut" declaration. Pointcuts are typically defined in aspects. An "advice" is a piece of code that is executed at each join point defined in a pointcut. There are three kinds of advice: before advice, around advice and after advice. Pointcuts and advice are encapsulated in an aspect. An "aspect" is like a class that encapsulates pointcuts and advices. An aspect can also include methods and field and can extend another class or aspect.

Please replace the paragraph on Page 12, lines 7-14, with the following amended paragraph:

One cannot create instances of aspects or type program variables as aspect types in ~~AspectJ~~ ASPECTJ. In other words, aspects are not first-class types in ~~AspectJ~~ ASPECTJ. Also, there is no notion of sub-type relation among aspects, although one aspect can extend another aspect. The semantics of aspect inheritance is not same as for classes. When one aspect extends another aspect it does not override pointcuts or advices defined in the parent aspect. In ~~AspectJ~~ ASPECTJ, first the child pointcut is executed and then the parent pointcut is executed. For instance, one may consider the Observer pattern (see also *supra*). [4] The following is a code snippet of the Notify aspect:

Please replace the paragraph bridging Pages 13-14 with the following amended paragraph:

Both the ~~AspectJ~~ ASPECTJ and ~~HyperJ~~ HYPERJ approaches go beyond traditional OO concepts. The basic premise for both approaches is that modularity goes well beyond what can be expressed in traditional OO languages. Similar patterns of code fragments occur in many places (e.g. logging code fragment). ~~AspectJ~~ ASPECTJ achieves modularity by collecting all similar patterns into a single *aspect*, and provides rules for injecting those patterns into other classes as and when needed. Code injection happens at compilation time. ~~HyperJ~~ HYPERJ also follows a similar approach, but treats similar code patterns as a separate hyperslice and provides rules for composing hyperslices. Once

again, hyperslice composition happens at compilation time. In order to make ~~AspectJ~~ ASPECTJ more fluid or ~~HyperJ~~ HYPERJ more morphogenic, one needs to go beyond compile time compositions. [22] One needs to treat aspects in ~~AspectJ~~ ASPECTJ as first-class types, as well as hyperslice in ~~HyperJ~~ HYPERJ. Once one treats aspects and hyperslices as first-class types, one can then compose them dynamically as needed. Also, once one treats them as first-class types one can apply standard type analysis to ensure type-safe composition. Extension types are one way to achieve the above goals. In extension types , one can treat hyperslices and aspects as first-class types and by treating them as first-class types one increases the expressiveness of both ~~AspectJ~~ ASPECTJ and ~~HyperJ~~ HYPERJ approaches.

Please replace the paragraph on Page 15, lines 6-18 with the following amended paragraph:

Class-based languages, such as ~~Java~~ JAVA and C++, provide a one-dimensional view of a class hierarchy; that is, two classes are related (through sub-class relation) only if one of them is an ancestor of the other in the class hierarchy. The control point (i.e., the sub-type relation) is closely tied to the representation (i.e., the class hierarchy). The close tie between sub-type relation and class hierarchy has been debated elsewhere in other contexts (this will be treated later). ~~HyperJ~~ HYPERJ goes one step further and views a class hierarchy as one hyperslice and provides composition operations to compose several such hyperslices to create a new class hierarchy that somehow captures

all the features of the individual hyperslices. What is missing in ~~HyperJ~~ HYPERJ is the semantic relation (i.e., control points) between the new class and the old hyperslices. Rather than explicitly create a new class for every set of composition operation and (possibly) discard the old hyperslices, a new composed type (extension type) is now contemplated which establishes a sub-type relation between the new type and its constituent types.

Please replace the paragraph on Page 17, lines 9-14 with the following amended paragraph:

With regard to method dispatch, let P be the declared type of p and Q be the run-time type of an object o pointed by p. A method lookup $p.m()$ in ~~Java~~ JAVA involves walking up the class hierarchy from Q and finding the closest implementation of m. In an extension type one can define different kinds of method dispatches. Let α be the extension type of a variable p and let β be the runtime extension type of the object pointed by p, so that $\beta <: \alpha$. With this in mind:

Please replace the paragraph on Page 23, lines 1-2 with the following amended paragraph:

The disclosure now turns to how extension types can be used in AOSD. First, the ~~HyperJ~~ HYPERJ approach will be focused upon, and thence the ~~AspectJ~~ ASPECTJ approach.

Please replace the paragraph on Page 25, lines 10-14, with the following amended paragraph:

Turning to ~~AspectJ~~ ASPECTJ and extension types, ~~AspectJ~~ ASPECTJ (as described in [6]) supports what Kiczales refers to as static AOP [22]. In [6], the aspects and classes are relatively fixed or static in that changing them involves editing the program. Fluid AOP will allow the easy remodularization of both aspects and classes. This dynamic remodularization is related to dynamic classification.

Please replace the paragraph on Page 26, lines 7-8, with the following amended paragraph:

Now one can extend the Notify aspect (similar to what is done in ~~AspectJ~~ ASPECTJ) to create a new aspect:

Please replace the paragraph on Page 27, lines 1-4, with the following amended paragraph:

One may note that the extension type (Employee, Notify) contains both Employee class and Notify aspects. In the context of ~~AspectJ~~ ASPECTJ one can define an "aspect extension type" to include both classes and aspects as elements. In general, an aspect extension type can be defined as follows:

Please replace the paragraph bridging Pages 29-30 with the following amended paragraph:

A Visitor pattern allows one to add a new operation to a class hierarchy without modifying the classes in the class hierarchy. Traditionally, in languages like ~~Java~~ JAVA, a Visitor pattern is implemented using a double-dispatch mechanism. Using ~~HyperJ~~ HYPERJ or ~~AspectJ~~ ASPECTJ, one can avoid such a double-dispatch mechanism. Herebelow, it will be shown that by using parameterized extension type one can implement a type-safe Visitor pattern using a single dispatch mechanism.

Please replace the paragraph on Page 35, lines 1-7, with the following amended paragraph:

Consider the Observer pattern discussed earlier. In languages like ~~Java~~ JAVA that do not support multiple inheritance of classes, Employee will be a sub-class (i.e., sub-type) of the class Subject. But conceptually it is known that Employee is not a Subject. In other words, there is no conceptual relation between the Employee class and the Subject class. The Employee class really does not care about attach() and detach() method, and it only requires the notify() method (not even how the notify() method is actually implemented).

Please replace the paragraph on Page 39, lines 11-19, with the following amended paragraph:

Mezini and Ostermann use concepts of family polymorphism in Caesar. [20]. Their main goal is to extend ~~AspectJ~~ ASPECTJ approach to allow a more flexible reuse and componentization of aspects. Family polymorphism allows one to group a set of classes to participate in collaboration. [21]. Interestingly one can use virtual types to provide a type safe family extension. Extension types are related to family polymorphism; the set of elements in an extension type belong to the same family type. Unlike Caesar or other similar approaches one can mix and match elements to create family types on-the-fly. On the other hand family classes are statically created in Caesar. Also, the semantics of Caesar family class is quite different compared to the semantics of extension types.

Please replace the paragraph on Page 40, lines 14-18, with the following amended paragraph:

In recapitulation, there have been introduced herein extension types for variational modeling. It has been demonstrated how extension types help simplify many concepts in AOSD and design patterns. Extension types is a simple way to implement multiple and dynamic classifications. Future steps could involve providing static and dynamic semantics for extension types, and to implement extension types in ~~Java~~ JAVA.

Please replace the paragraph on Page 43, line 4, with the following amended paragraph:

[8] http:// colon slash slash www. dot research. dot ibm. dot com/ slash hyperspace/ slash HyperJ/ slash HyperJ. dot htm

Please replace the paragraph on Page 44, lines 13-14, with the following amended paragraph:

[19] M. Fowler. Dealing with roles. http:// colon slash slash www. dot Martinfowler.~~dot~~ com/ slash apsupp/ slash roles. dot pdf, July 1997.

Please replace the paragraph on Page 45, line 9, with the following amended paragraph:

[24] D. Lea http~~:~~// colon slash slash gee~~.~~ dot cs~~.~~ dot oswego~~.~~ dot edu~~/~~ slash dl~~/~~ slash papers~~/~~ slash groups~~.~~ dot pdf